Communication and Timing Issues with MPI Virtualization

Alexandr Nigay, Lukas Mosimann, Timo Schneider, Torsten Hoefler
Department of Computer Science
ETH Zurich
Switzerland
{alexandr.nigay,mosimann,timos,htor}@inf.ethz.ch

ABSTRACT

Computation–communication overlap and good load balance are features central to high performance of parallel programs. Unfortunately, achieving them with MPI requires considerably increasing the complexity of user code. Our work contributes to the alternative solution to this problem: using a *virtualized* MPI implementation. *Virtualized* MPI implementations diverge from traditional MPI implementations in that they map MPI processes to user-level threads instead of operating-system processes and launch more of them than there are CPU cores in the system. They are capable of providing automatic computation–communication overlap and load balance with little to no changes to pre-existing MPI user code. Our work has uncovered new insights into MPI *virtualization*: Two new kinds of timers are needed: an MPI-process timer and a CPU-core timer, the same discussion also applies to performance counters and the MPI profiling interface. We also observe an interplay between the degree of CPU oversubscription and the rendezvous communication protocol: we find that the intuitive expectation of only two MPI processes per CPU core being enough to achieve full computation–communication overlap is wrong for the rendezvous protocol—instead, three MPI processes per CPU core are required in that case. Our findings are expected to be applicable to all *virtualized* MPI implementations as well as to general tasking runtime systems.

ACM Reference Format:
Alexandr Nigay, Lukas Mosimann, Timo Schneider, Torsten Hoefler. 2020. Communication and Timing Issues with MPI Virtualization. In *27th European MPI Users' Group Meeting (EuroMPI/USA '20), September 21–24, 2020, Austin, TX, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3416315.3416317

1 INTRODUCTION

Performance of parallel programs depends on many factors, among which are the computation–communication overlap and the load balance. Overlapping communication with computation can hide network latency and thus lead to better performance. Parallel computations split the overall volume of computation into many parts and assign them to individual processors. This load must be carefully balanced across the processors, so that the program does not have to wait for some processors while other processors are idle.

Unfortunately, computation–communication overlap and good load balance are non-trivial to achieve in MPI [18]. A traditional MPI implementation maps each MPI process, each of which gets assigned a share of the computation, to an operating-system process and usually launches one MPI process per core; oversubscription is possible with the popular MPI implementations, but is rarely used in practice. Achieving computation–communication overlap with such an MPI implementation requires using MPI's *nonblocking* communication calls which make code more complex and harder to read compared to code written with the conceptually-simpler *blocking* calls. Furthermore, even if nonblocking calls are used, the overlap may still require the user code to periodically yield control to the MPI library to allow it to progress the network operations, which complicates the code even further [11]. As for the load balancing, it can be implemented with MPI by explicitly exchanging work, but this, again, complicates the user code.

In practice many MPI programs actually use fewer MPI processes than available cores, parallelizing computation in two ways: a coarse-grained parallelization is done using MPI and a fine-grained parallelization uses e.g., OpenMP to make use of the remaining cores. This again complicates user code. On the contrary in this work we assume only MPI is used, and MPI is initialized using MPI_THREAD_SINGLE, i.e. each MPI process consists of one thread.

MPI *virtualization* provides a way of achieving overlap and load balance while keeping the user code conceptually simple. In contrast to a traditional MPI implementation, a virtualized MPI implementation maps each MPI process to a user-level thread (ULT) and launches more of them than there are CPU cores in the system. Furthermore, as the number of MPI processes increases, the amount of work per process tends to decrease, making the user code transfer the control to MPI more frequently, alleviating the problem with progressing the network operations.

Multiple researchers have presented *virtualized* MPI implementations: Adaptive MPI (AMPI) [12], MPC-MPI [22], FG-MPI [16], TMPI [27], Toucan [17] (it adheres to a different programming model but is otherwise close in spirit to MPI virtualization), and TOMPI [5].

The goal of this study is to contribute to the understanding of MPI virtualization. We have discovered a need for new time-measurement calls: to measure time from the point of view of an MPI process and from the point of view of a CPU core. These timers complement the capabilities of MPI's wall-clock timer, `MPI_Wtime`. On the communication side, we have found the interplay between

EuroMPI/USA '20, September 21–24, 2020, Austin, TX, USA

ACM ISBN 978-1-4503-8880-1/20/09…$15.00
https://doi.org/10.1145/3416315.3416317

the degree of oversubscription and the MPI rendezvous communication protocol [30]. Contrary to intuition, having only two MPI processes per CPU core would not be sufficient to achieve full computation–communication overlap if a three-way rendezvous protocol is used. We present our findings after a deeper discussion of MPI virtualization.

2 MPI VIRTUALIZATION

The idea of using oversubscription to hide latency of a time-consuming operation is an old and widely-used technique outside of MPI. The MPI virtualization idea applies this technique in the context of implementing MPI—suspend an MPI process blocked in a communication call and switch to a process ready to continue executing user code outside of MPI. In the context of this work, MPI virtualization refers specifically to the practice of mapping MPI processes to user-level threads, as opposed to mapping them to OS-level processes.

Oversubscribing the CPU is technically possible with traditional MPI implementations, in which MPI processes are mapped to OS-level processes. However, mapping MPI processes to user-level threads provides multiple advantages. Context switches between ULTs are faster than context switches between OS-level processes, because it is possible to do without involving the operating system. This full control over a MPI processes's state also provides for simpler migration of MPI processes between cores and nodes, and it also provides for simpler checkpointing, both demonstrated by Adaptive MPI [12, 13]. MPI virtualization enables the reduction in the memory consumption associated with shared-memory communications—single copy from the source buffer directly to the destination buffer is possible without special kernel modules, since all MPI processes on a node can share the address space of the parent OS-level process—which is demonstrated by MPC-MPI [21]. MPC-MPI also shows that MPI virtualization can enable better interoperation between MPI and OpenMP [2].

A virtualized MPI implementation takes on a part of responsibilities of the operating system: context switching and scheduling. It effectively *virtualizes* the CPU in the same way as an operating system virtualizes it through multitasking.

However, implementing MPI processes as user-level threads does have its drawbacks: managing contexts from user space requires the ULTs to be spawned off the same OS-level process, meaning that each MPI process no longer has its own private address space, which is the standard assumption of MPI programs. Specifically, it is normal for an MPI program to assume that each MPI process has its own private set of global variables. Such a program would behave incorrectly when used with a virtualized MPI implementation if any of the global variables assume different values across different processes over the course of the program's execution, thus necessitating the privatization of global variables in user code [19, 31].

2.1 Scheduling of virtualized MPI processes

Each process in a virtualized MPI implementation can be in one of the following four *states*:

(1) **`INIT`**: when the process has not yet left the MPI initialization call, `MPI_Init`; all processes start in this state;

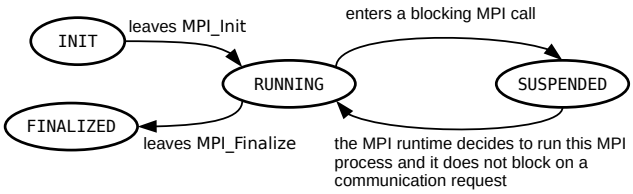

Figure 1: The process state transition diagram of a virtualized MPI implementation. Each MPI process resides in one of these four states at any moment.

(2) **`RUNNING`**: when the process is currently being executed by a CPU core. If the underlying operating system decides to suspend the OS-level process hosting the MPI process, that MPI process is still considered to be in the `RUNNING` state, because it was not the decision of the MPI runtime to suspend the process, and these states only reflect the point of view of the MPI runtime; on each CPU core, at most one MPI process can be in the `RUNNING` state at any moment;
(3) **`SUSPENDED`**: while the MPI process is not being executed by the CPU core as per the MPI runtime's decision;
(4) **`FINALIZED`**: when the MPI process has returned from `MPI_Finalize`; this is the terminal state.

MPI processes transition between states according to the diagram presented in Figure 1.

On *scheduling points*, the MPI runtime has the power to change the state of the currently-running MPI process to `SUSPENDED` and transition another MPI process to the `RUNNING` state. Existing virtualized MPI implementations use cooperative multitasking and place the scheduling points into blocking MPI calls, while some also provide means for the MPI processes to explicitly place a scheduling point by yielding to the scheduler.

2.2 Implementing a virtualized MPI

MPI processes can be mapped to user-level threads through different approaches. Adaptive MPI [12] uses the abstractions of the Charm++ runtime [14, 15]. MPC-MPI uses a custom implementation of user-level threads [21].

Our virtualized MPI implementation, named TinyMPI[1], which we use as the research vehicle of this work, uses a custom implementation of user-level threads: it launches an OS-level process on each compute node (one shared-memory domain), spawns an OS-level thread per each CPU core on the node, and, for each of those, spawns multiple user-level threads, each corresponding to an MPI process.

TinyMPI's context switching mechanism is analogous to that of Grappa [20]: it switches the stacks and callee-saved registers, according to the calling convention of the platform, of the to-be-suspended MPI process and of the to-be-resumed MPI process. This essentially makes a context switch equivalent to a function call.

Before presenting the findings of our work, we briefly cover the work related to MPI virtualization.

[1] http://spcl.inf.ethz.ch/Research/Parallel_Programming/TinyMPI/

2.3 Work related to MPI virtualization

Other paradigms of MPI implementation and general tasking systems have targeted the same benefits as the MPI virtualization paradigm does.

2.3.1 Overlap and load balancing. Many tasking runtime systems target computation-communication overlap and load-balancing but outside of the scope of MPI.

Charm++ [14] is a parallel programming system which oversubscribes CPUs with a large number of *chares*. Chares interact by sending messages to each other and get invoked to perform computation in response to an incoming message. Charm++ can migrate chares across cores and nodes, enabling automatic load balancing. Charm++ hides the communication latency by scheduling a runnable chare in place of a chare waiting for communication. Adaptive MPI [12], a virtualized MPI implementation, is based on Charm++.

Grappa [20] is a system which aims to simplify the programming of irregular applications by providing automatic network-latency hiding through oversubscription and load balancing via task migration, among other features. Grappa's programming model is also centered around splitting the application logic into a set of concurrently-executed, communicating tasks. To enable the performance benefits it provides, Grappa implements the tasks as user-level threads.

Dang et al. [3] describe an MPI implementation which, by dropping the support for the wildcard semantics of MPI, can sustain thousands of threads on the same node communicating via MPI. This system uses ULTs, as virtualized MPI implementations do, but it does not map MPI processes to those threads; instead, MPI processes are multithreaded. The work aims to lift the performance issues associated with calling MPI from multiple threads concurrently, a technique which, among other benefits, enables hiding the communication latency.

2.3.2 Single-copy shared-memory communication. Hybrid MPI [7] aims to provide single-copy communication between MPI processes residing on the same compute node. Hybrid MPI maps MPI processes to OS-level processes, as in traditional MPI implementations, but these processes share the heap portion of their address space through OS-provided mechanisms, which enables Hybrid MPI to implement single-copy shared-memory communications.

KNEM [9] is a Linux kernel module which enables single-copy shared-memory communication. It is designed to work with traditional MPI implementations and was integrated into the mainstream MPI implementations, MPICH2 and Open MPI. It provides single-copy communication by leveraging the fact that the OS kernel can see the address spaces of all the OS-processes and can thus perform a copy from the sender's buffer directly to the receiver's buffer. Recent Linux versions have integrated similar functionality into the mainline kernel [28].

Buntinas et al. [1] summarize different options of implementing shared-memory communication in MPI, including single-copy mechanisms: kernel-module support, OS mechanisms intended for debuggers, and the use of RDMA capabilities of network interfaces to loop the data back from one process to another on the same network node.

We now proceed to discussing the new insights into MPI virtualization which we have uncovered in this work.

3 ISSUES WITH TIME MEASUREMENT

MPI provides a portable and convenient way to measure time: the `MPI_Wtime` call, which "returns a floating-point number of seconds representing elapsed *wall-clock* time since some time in the past" (emphasis ours), as the text of the specification puts it [18]. When MPI virtualization is introduced, the facilities provided by `MPI_Wtime` are no longer sufficient and have to be extended, as we will explain in this section.

Consider the following code:

```
int main() {
    initialize();
    /* A */
    for(int i = 0; i < ITERS; i++) {
        compute();
        MPI_Sendrecv(...);
    }
    /* B */
    finalize();
}
```

Assume that we want to find the answers to the following two questions about the code above:

- How much time has *each MPI process* spent in the section of code between points **A** and **B**? We call such a measurement a "process-POV"[2] measurement.
- How much time has *each CPU core* spent in the section of code between points **A** and **B**? We call such a measurement a "core-POV" measurement.

With a traditional MPI implementation these two questions are the same because each MPI process is mapped to an OS-level process and will most likely be pinned to a particular CPU core. The answer to both of these questions will then be found by adding two **`MPI_Wtime`** calls, one at **A** and one at **B**, then subtracting the value returned by the first from the value returned by the second call. However, with a virtualized MPI implementation, those two questions are distinct, and the presented solution will yield incorrect measurements for both of them. We will now discuss each case individually.

3.1 Why process-POV measurements are needed

We define process-POV measurements to be drawn from a clock which ticks only while the requesting MPI process is in the `RUNNING` state and not while it is in any of the other states, as per the definition of states given in Section 2.1.

Process-POV measurements are necessary when the user code needs to measure performance from the point of view of an MPI process. Process-POV measurements are also necessary for load-balance studies, where MPI processes are expected to spend different amounts of time in the same section of code—in this case it is necessary to measure the time accumulated by each process individually.

If several MPI processes share a CPU core, then wrapping a code section with a pair of `MPI_Wtime` calls and taking the readings on

[2] "POV" standing for "point of view"

Table 1: Output of the existing virtualized MPI implementations for the code in Listing 1—`MPI_Wtime` calls return misleading values.

MPI	Task topology (#p/#c)	Value of `elapsed`		Total execution time
		proc 0	proc 1	
Open MPI (no virtualization)	2 / 2	30	30	30
AMPI	2 / 1	**60**	30	60
MPC-MPI	2 / 1	**60**	30	60
TinyMPI `MPI_Wtime`	2 / 1	**60**	30	60
TinyMPI `MPIX_Rtime`	2 / 1	30	30	60

Listing 1: Outline of the code exposing the misleading `MPI_Wtime` readings returned by virtualized MPI implementations.

```
int main() {
    MPI_Init();
    double start = MPI_Wtime();
    sleep(/* 30 seconds */);
    MPI_Barrier();
    double elapsed = MPI_Wtime() - start;
    printf("%f\n", elapsed);
    MPI_Finalize();
}
```

one of those MPI processes will include the periods during which the process was in the SUSPENDED state and will thus yield incorrect readings. This is visually demonstrated in the topmost portion of Figure 2. The goal is to count the time that a MPI process spent between points A and B but only while it was in the RUNNING state, which is similar in spirit to the "CPU time" measured for processes by the operating system.

To demonstrate that existing virtualized MPI implementations cannot provide process-POV timings without relying on external-to-MPI timing tools, we execute the code in Listing 1 with AMPI (Charm++ 6.8.2), MPC-MPI 2.5.0, Open MPI 3.0.0, and TinyMPI. The code is executed with two MPI processes co-located on the same CPU core, except for Open MPI, which is not a virtualized MPI and uses two CPU cores. Table 1 presents the output of each MPI implementation. Following the standard-mandated semantics, the `MPI_Wtime` calls return the value of 30 (rounded to full seconds, as accuracy is not of concern here but semantics) with the non-virtualized Open MPI. In contrast, one of the processes with all tested virtualized MPI implementations measures 60 seconds instead of 30, except for TinyMPI with `MPI_Wtime` replaced with `MPIX_Rtime`, the process-POV timer call that we propose. Such measurements suggest that the two processes have collectively worked for 90 seconds, where in reality it was 60. The discrepancy is caused by the fact that the timing calls include the periods of time during which the calling processes was suspended. Thus, `MPI_Wtime` cannot provide process-POV measurements under MPI virtualization.

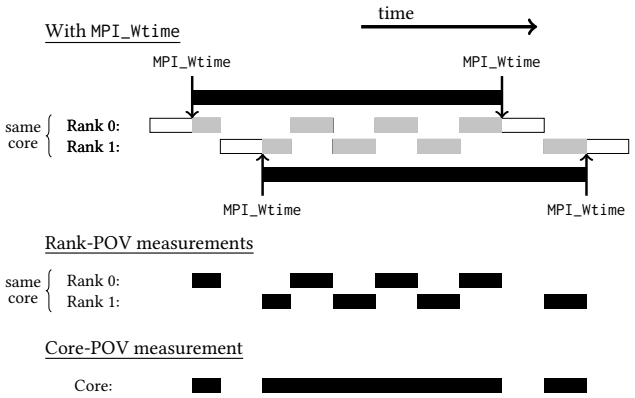

Figure 2: The difference between time measurements from an MPI process point of view ("process-POV"), from a core's point of view ("core-POV"), and the measurements returned by `MPI_Wtime`. The gray shading in the top portion of the figure represents process-POV measurements, which are shown separately in the middle portion.

These misleading time measurements cause benchmark applications that rely on `MPI_Wtime` to inadvertently self-report erroneous performance figures when used with a virtualized MPI implementation, which we observe with the HPCCG benchmark from the Mantevo suite [10]. We run HPCCG with TinyMPI and MPC-MPI 2.5.0 using four MPI processes in total, all of them sharing one CPU core. Unfortunately, we could not successfully run this experiment with AMPI (Charm++ 6.8.2): the code runs with four processes on four CPU cores but hangs with four processes on one core. Table 2 presents the performance numbers self-reported by HPCCG in these runs. The full running time was measured by external timers to be 11.5 seconds for TinyMPI and 12 seconds for MPC-MPI. For both TinyMPI and MPC-MPI, the wall-clock `MPI_Wtime` reports values almost equal to that full running time, which is of four processes sharing the same core, but this measurement is supposed to be made for a single process. This leads to incorrectly high time measurements, which in turn lead to incorrectly low calculated MFLOP/s values—HPCCG calculates this metric by dividing the number of floating-point operations performed *by a single rank* by the time taken by that *single rank*. The process-POV `MPIX_Rtime`, the call which we discuss in detail subsequently, correctly accumulates the time spent by the calling MPI process only, resulting in realistic time measurements and performance figures.

Rodrigues et al. [25] encounter a similar effect in their analysis: the authors have observed wide fluctuations in the timestep execution time measured by a single MPI process sharing the same CPU core with 15 other processes, which was attributed to unpredictable interleaving of those 16 processes by the virtualized MPI runtime. Based on the description of the observation, we conjecture that process-POV timers would have removed the fluctuations and yielded stable measurements.

Let us now describe how process-POV time measurements may be implemented in a virtualized MPI implementation.

Table 2: HPCCG inadvertently reports misleading performance numbers when using a virtualized MPI implementation.

	MPC-MPI `MPI_Wtime`	TinyMPI `MPI_Wtime`	TinyMPI `MPIX_Rtime`
Tot. time [s]	11.1	10.9	2.85
DDOT time [s]	3.8	4.9	0.16
Tot. MFLOP/s	1755	1792	6846
DDOT MFLOP/s	319	245	7513

HPCCG reports measurements for rank 0. In these runs, 4 MPI processes share the same CPU core and collectively take 11.5 seconds for TinyMPI and 12 seconds for MPC-MPI.

Listing 2: Our implementation of the process-POV `MPIX_Rtime`. In all statements, `R` and `T` refer to the current process's fields. Function `now()` returns current wall-time.

```
// There is one instance of a process_local
// variable for each process.
proc_local timestamp R; //process-POV clock
proc_local timestamp T; //helper

on_proc_init():  // Called for each proc in MPI_Init.
    R = 0;
    T = now();

on_proc_suspend(): // Advance the process-POV clock.
    R += now() - T;

on_proc_resume():
    // The process-POV time did not change.
    // Only the helper must update.
    T = now();

MPIX_Rtime():
    // Advance the clock and return the value.
    R += now() - T;
    T = now();
    return R;
```

3.2 Obtaining process-POV Time Measurements

In order to obtain process-POV measurements for a particular MPI process, we must use a clock which is ticking only while that process is kept in the `RUNNING` state by the MPI runtime. A wall-clock timer cannot directly serve this purpose, because it ticks constantly, without regard to the MPI runtime's user-space scheduling of MPI processes.

The process-POV timer is only required when the timed section of code contains a virtualized MPI's *scheduling point*. If the code section does not contain any such points and hence the MPI process does not get suspended by the MPI runtime while inside of it, then `MPI_Wtime` can be used, since no periods during which the calling process was `SUSPENDED` will be captured.

Since process-POV time measurements complement rather than replace the wall-clock time measurements, we suggest introducing a new timer call into the MPI standard: `MPIX_Rtime`, named so after "Rank-POV"; following the convention, the `X` in the prefix `MPIX` signifies "extension". The new call is defined as returning a floating-point number of seconds elapsed since a fixed time in the past but only including the time during which the *calling* process was `RUNNING`.

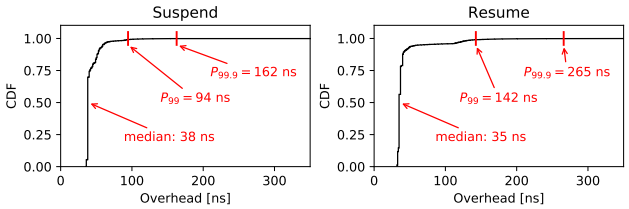

Figure 3: Cummulative distribution function of the per-task-switch overhead of the proposed process-POV timer logic. P_{99} and $P_{99.9}$ denote 99th and 99.9th percentiles.

Listing 2 outlines our implementation of `MPIX_Rtime`. It does not rely on specific hardware support. Each MPI process internally holds two values: `R`—the process's own point-of-view clock, and `T`—a helper variable. These two variables can be stored in the control data structure maintained for each process by the MPI runtime. The value of `R` appears to be incrementing only while its owning process is `RUNNING`, hence it can feed `MPIX_Rtime`.

This implementation is robust to MPI process migration: if a process migrates to a core whose timer is not synchronized with that of the previous core, its process-POV timekeeping will still be correct because the wall-clock timestamps from different cores are never related to each other in any way—only the values from the same core are used in the subtractions.

The on-suspend and on-resume parts of the process-POV logic have to be executed on each MPI process context switch, which is potentially a frequent event; therefore it is important to evaluate their overhead. Figure 3 presents the statistical summary of absolute time measurements of approximately 70,000 invocations of on-suspend and on-resume logic implemented in C++ on a 2.2 GHz Intel Xeon E5-2660V2. The 99th percentile is below 100 ns for the on-suspend logic and is below 150 ns for the on-resume logic.

3.3 Why core-POV measurements are needed

Core-POV measurements are necessary for load-balance studies in which the imbalance occurs due to CPU cores dynamically changing their frequency, making the same code section take different amounts of time on different cores. The "Core-POV" portion of Figure 2 visually presents the difference between core-POV and process-POV measurements.

The wall-clock `MPI_Wtime` timer cannot fulfill the task of counting core-POV time. It is definitely possible to arrange for the first process entering the code section to start the timer and the last process leaving it to stop the timer. However, such measurements would be inaccurate, because sections of code outside of the target section will also be included—all other MPI processes sharing the same CPU core will be unblocking at the last scheduling point, which will always be *before* the target code section, and that section of code before the target section will get included in the measured period. The same applies to the section of code *after* the target section up to the next scheduling point. The core-POV measurement will thus be overestimated. The degree of inaccuracy directly depends on how long the leading and the trailing sections of code

take: if those take a small amount of time relative to the target section, then the inaccuracy may be acceptable; otherwise, a different approach is needed.

Rodrigues et al. [25] observe this situation but in the context of performance counters: in their cache-utilization analysis of a code running with AMPI, the authors aim to count cache misses on each core when executing a specific code section. The authors use the approach of having the first-entering rank start the counters and the last-leaving rank stop the counters. Core-POV instrumentation logic would have provided the desired measurements without the described inaccuracy.

Rank-POV timers can fulfill the task of measuring core-POV time only partially—the measurements will be correct only if the MPI processes do not migrate between cores, in which case summing up process-POV measurements for the target code section across all MPI processes on that core will yield the correct core-POV measurement. This can be implemented using the current MPI standard, e.g., by using `MPI_Get_processor_name` to identify a CPU core, then calling `MPI_Comm_split` to create a communicator per each core, making an `MPI_Reduce` call on each core to sum the process-POV values, and finally having one MPI process per each core report the core-POV value. However, this sequence appears to be too complex for the simple goal of timing a section of code in a core-POV manner, so support from MPI is desirable.

Conversely, if MPI processes are allowed to migrate between cores during the program run, then summing process-POV values will yield a wrong result—a process will incorrectly attribute all the amount of time it has accumulated in the measured code section to the core on which it happened to be executing at the moment of the summation, while all the other cores which have executed any portion of the measured section on this process's behalf will get attributed nothing. In this case, the support from the MPI runtime is not just desirable but necessary.

3.4 Obtaining Core-POV Time Measurements

The crux of the problem with obtaining core-POV time measurements is not that we need a special kind of core-POV clock—any standard wall-time clock is already a core-POV clock—but rather that the timer has to be started and stopped at specific events: the timer has to be started when any MPI process on this core enters the measured code section or resumes execution at a scheduling point inside that section, and it has to be stopped when any MPI process leaves the section or gets suspended in a scheduling point within it.

To provide such functionality from MPI, we suggest extending the MPI interface with three new calls:

```
void MPIX_Start_processor_timer()
void MPIX_Stop_processor_timer()
double MPIX_Ptime()
```

The first two calls are to be used to demarcate the section of code for which core-POV time measurements are desired to be taken. The `MPIX_Ptime` call, named after "Processor time", is used to retrieve the measurement; it provides access to the same core-POV timer to all MPI processes sharing a core.

Listing 3 outlines our proposed way of implementing these core-POV timer calls. The information on whether or not a particular MPI process is currently executing within the core-POV-timed section of code is inherent to that process, because it depends on where in the program that process is executing. Thus, whether or not the time accumulation should take place depends on the process state. The MPI process-local variable `M` is used for that purpose, with the special value `-1` indicating that no core-POV timer is active and a non-negative value indicating that the timer is active. This variable can be placed into the control data structure maintained for each process by the MPI runtime. On the other hand, the information on how much time a certain core has accumulated in the timed section of the code is inherent to that core. Hence, each core maintains its private running sum in the core-local variable `C`, which can be stored in the control data structure kept for each CPU core by the MPI runtime.

Listing 3: Our implementation of core-POV timers. In all statements, `M` and `C` refer to the current process's and current core's fields respectively. Function `now()` returns current, non-negative wall-time as measured by a call such as `rdtsc` or `gettimeofday`.

```
// There is one instance of a core_local
// variable for each CPU core.
core_local double C; // The core-POV timer.
proc_local double M; // A per-proc helper variable.

on_core_init(): // once per core in MPI_Init
    C = 0;

on_proc_init(): // once per proc in MPI_Init
    M = -1;

MPIX_Start_processor_timer():
    M = now();

on_proc_suspend():
    //if in the metered section, add to timer
    if (M > 0): C += now() - M;

on_proc_resume():
    //if in the metered section, refresh helper
    if (M > 0): M = now();

MPIX_Stop_processor_timer():
    //Accumulate contributions
    if (M > 0): C += now() - M;
    //leave the metered section
    M = -1;

MPIX_Ptime():
    return C
```

As with the process-POV timer logic, the on-suspend and on-resume parts of the core-POV timer logic have to be executed on each MPI process switch, which is potentially a frequent event. Figure 4 presents the statistical summary of absolute time measurements of approximately 70,000 invocations of on-suspend and on-resume logic implemented in C++ on a 2.2 GHz Intel Xeon E5-2660V2. The 99th percentile is at 170 ns for the on-suspend logic and is below 150 ns for the on-resume logic.

3.5 Discussion of process-POV and core-POV timers

The time measurement issues described in this section may potentially be resolved by using instrumentation external to MPI, e.g., by

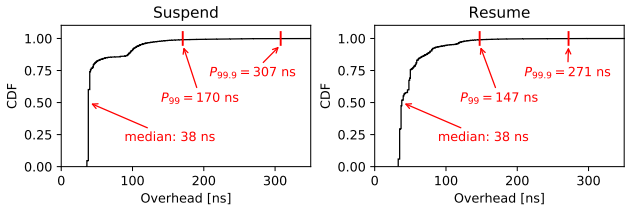

Figure 4: Cummulative distribution function of the per-task-switch overhead of the proposed core-POV timer logic. P_{99} and $P_{99.9}$ denote 99th and 99.9th percentiles.

using timing facilities provided by a particular virtualized MPI implementation: Adaptive MPI, for example, provides a performance-analysis tool called *Projections* [8]. However, a portable MPI program would confine itself to using only the facilities included in the MPI standard. Hence, we argue the MPI standard should include virtualization-aware time-measurement features.

MPI simulation frameworks—such as SMPI [4], xSim [6], MPI-SIM [23], and the MPI simulator described in [24]—partially share the timing issues with virtualized MPI implementations. Such frameworks aim to use one system to simulate execution of an MPI application on another, possibly larger, system. In such circumstances, `MPI_Wtime` has to be altered to return "virtual time", which could be similar to the process-POV time returned by `MPIX_Rtime` proposed in our work. However, the two functions would have different meanings: a simulator's `MPI_Wtime` would be returning wall-clock timer's readings on a hypothetical target system, which is different from the real system executing the simulation; whereas `MPIX_Rtime` would be returning process-POV reading as seen by the same system that executes the calculation.

3.5.1 Performance counters and the MPI profiling interface. We presented the process-POV and core-POV measurement problem in the context of time measurement; however, measurements with performance counters are facing the exact same problem: process-POV and core-POV logic is needed in order to obtain meaningful readings. The MPI profiling interface [18] faces the same problem as well: if a profiler implementation aims at gathering statistics for MPI calls containing scheduling points, then it must employ process-POV and/or core-POV measurement mechanisms.

Thus, the introduction of virtualization into MPI brings the need to have new kinds of timer calls: for measuring time from the point of view of an MPI process and from the point of view of a CPU core. The underlying reason for introducing both kinds of timers is the fact that several MPI processes share the same CPU core. We will now discuss another insight that this work has uncovered.

4 RELATION BETWEEN VIRTUALIZATION RATIO AND MPI'S COMMUNICATION PROTOCOL

MPI virtualization involves launching more than one MPI process per CPU core; the exact number of MPI processes per CPU core is referred to as the *virtualization ratio* by the authors of Adaptive MPI [12], we adopt this terminology. The choice of the virtualization ratio has an unexpected link to the MPI rendezvous communication protocol. To see this, we need to inspect the computation–communication overlap achieved by a virtualized MPI implementation for an application that does not implement overlapping on its own. We will first introduce our methodology of measuring the achieved overlap.

Listing 4: Outline of the microbenchmark employed for evaluating achieved overlap. The code mimics a 2-dimensional stencil application. Each MPI process communicates with 4 neighbors.

```
int main(int argc, char** argv) {
  MPI_Init();
  for (int i = 0; i < ITERS; i++) {
    compute();  /*Comp. phase */

    /* Communication phase */
    MPI_Irecv(); MPI_Irecv();
    MPI_Irecv(); MPI_Irecv();
    MPI_Isend(); MPI_Isend();
    MPI_Isend(); MPI_Isend();
    MPI_Waitall();
  }
  MPI_Finalize();
}
```

4.1 Measuring the overlap

We use a microbenchmark which mimics a 2-dimensional stencil calculation with a computation phase and a blocking halo-exchange phase; its outline is presented in Listing 4. We build two controls into the benchmark. First, it is possible to enable and disable any of the two phases independently. Second, it is possible to freely adjust the amounts of both computation and communication.

The microbenchmark does not attempt to overlap computation and communication on its own: it uses the nonblocking MPI calls, but no computation is put between the initiation calls and the `MPI_Waitall`. Thus, any achieved overlap is entirely attributed to the underlying MPI implementation.

To measure the achieved overlap, we first run the microbenchmark with only the computation phase enabled and measure the time taken, which we refer to as T_{comp}; we then repeat the runs with only the communication phase enabled and obtain the measurement T_{comm}; finally, we re-run the code with both components enabled and obtain the overall measurement T_{full}. We then relate the three measurements:

- If $T_{full} = \max(T_{comp}, T_{comm})$, i.e., the shorter of the two components got completely hidden behind the longer, then the best possible overlap has been achieved;
- if $T_{full} = T_{comp} + T_{comm}$, then no overlap has been achieved—the two components merely got stacked on top of each other;
- if $\max(T_{comp}, T_{comm}) < T_{full} < (T_{comp} + T_{comm})$, then only partial overlap has been achieved.

Thus, a triplet of measurements—$\{T_{comp}, T_{comm}, T_{full}\}$—is used to evaluate the achieved overlap.

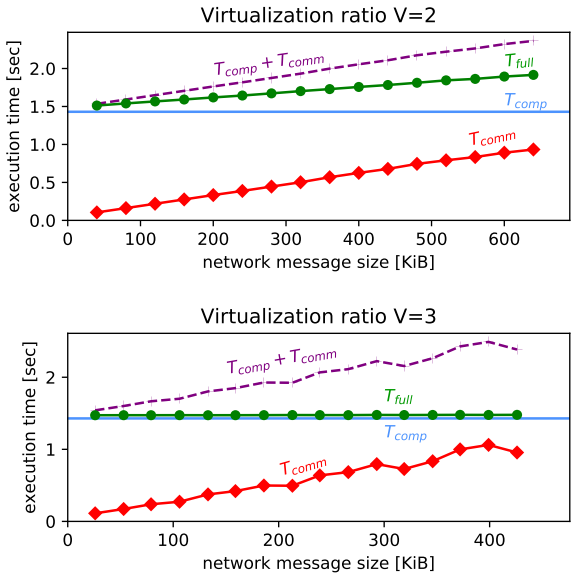

Figure 5: Overlap achieved by a microbenchmark with $V = 2$ (top) and $V = 3$ (bottom). The $V = 2$ configuration achieves only partial overlap, while the $V = 3$ configuration achieves full overlap.

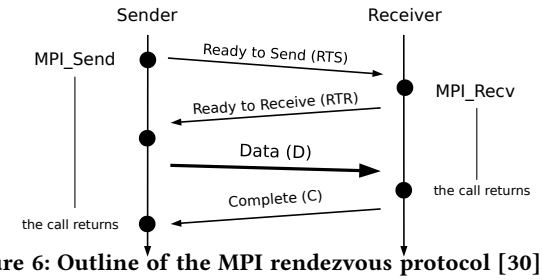

Figure 6: Outline of the MPI rendezvous protocol [30]. The sender and the receiver perform a handshake before starting the data transmission.

4.2 Rendezvous protocol needs at least three processes per core

We will use V to denote *virtualization ratio* where appropriate for brevity. Figure 5 presents the measurements from two series of runs of the overlap microbenchmark: with $V = 2$ in the top plot and $V = 3$ in the bottom plot. In these runs, the amount of computation is kept constant throughout the runs, so T_{comp} remains constant, but the amount of communication is varied, yielding a rising trend for T_{comm}. No shared-memory communication is performed during these runs—only the network communication is performed, which uses the rendezvous protocol [30] as outlined in Figure 6. The data for these plots was obtained using our virtualized MPI implementation, TinyMPI, running on two nodes of a cluster with 2.2 GHz Intel Xeon E5-2660V2 processors and a 56 Gbps InfiniBand FDR interconnect. The figure demonstrates that the achieved overlap improves considerably when going from $V = 2$ to $V = 3$, as seen from the fact that in the bottom plot the T_{full} line (the green line) lies directly on top of the T_{comp} line, which is the maximum of

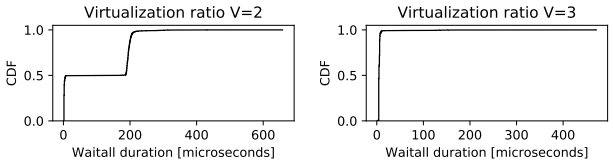

Figure 7: Cummulative distribution function of the amount of time spent by the overlap benchmark in each `MPI_Waitall` call. With $V = 2$ the distribution is strongly bimodal, while with $V = 3$ it is unimodal.

the two components. This data suggests that the rendezvous protocol requires at least three MPI processes per CPU core in order to achieve good computation–communication overlap. This finding contradicts the intuition which suggests that two MPI processes per core should be enough—when one blocks, the CPU switches to the other one.

Our analysis is based on the assumption that the rendezvous protocol is not offloaded from the CPU, i.e., the CPU has to take explicit action at each step of the protocol's handshake: to handle `ReadyToSend` and `ReadyToReceive` messages and initiate user data transfers.

With only one MPI process per core (i.e., $V = 1$), a rendezvous *send* operation leaves two idle gaps in CPU activity: while waiting for the `ReadyToReceive` and while waiting for the data transfer completion acknowledgment. During these waiting periods, the CPU core cannot progress the user's MPI program, since the only MPI process assigned to this core is `SUSPENDED`, waiting for its communication requests to be served; and the core is also unable to progress the communication operation, since it needs a reply from its peer.

With two MPI processes assigned to the same core ($V = 2$), one of these idle gaps is filled—the CPU core can switch to the other process. However, only one of the two idle gaps can be covered in this case—in the second idle gap, both processes are in the `SUSPENDED` state unable to become `RUNNING`. This means some Waitall calls will appear to finish faster than other (to the calling process). Overlap is achieved partially in this case.

Both idle gaps can only be covered if the core has three MPI processes to choose from ($V = 3$). In every of the idle gaps, there exists at least one MPI process which can transition to the `RUNNING` state, hence the CPU core is busy at all times. In this case, full computation–communication overlap is achieved.

Figure 7 presents the empirical validation of this explanation.

4.3 Discussion

We find evidence of a similar effect described by White [29], where the performance of PlasComCM is shown to increase sharply when changing from $V = 2$ to $V = 3$. We believe that our presented explanation applies to this case.

Our analysis is based on the assumption that the rendezvous protocol is executed by the CPU. If the protocol is entirely offloaded to the network hardware, then the full overlap may be achieved already with $V = 2$, because only one idle gap will be generated—the gap encompassing the entire handshake and the data transfer. However, offloading the rendezvous protocol to network hardware

was demonstrated to be non-trivial and requiring extensions even to such a feature-rich interface as Portals 4 [26]. MPI virtualization circumvents this problem by keeping the CPU responsible for progressing the protocol while still managing to achieve computation–communication overlap.

In our analysis we focus on the *sender* side of the rendezvous protocol, which generates two idle gaps in CPU activity. The *receiver* side of the rendezvous protocol generates either two idle gaps or one, depending on whether the receive was posted before or after the matching `ReadyToSend` has arrived. A single idle gap can be filled if $V = 2$; however, since each MPI process usually performs both sends and receives, the $V \geq 3$ recommendation still applies.

We focus on point-to-point communication in our analysis, but we expect these findings to apply to collective operations as well, if they employ rendezvous communication internally. While some MPI processes are blocked inside the collective, the processes which have not yet entered the collective or have already left it may proceed with computation. In this case, the overlap will happen between the communication triggered by the collective and the computation work done immediately before and after the collective call.

If `ReadyToReceive` arrives so shortly after the associated `ReadyToSend` has been sent that the MPI runtime is still inside the same invocation that triggered the `ReadyToSend`, and if the MPI implementation is capable of processing the `ReadyToReceive` immediately, then the sender side of rendezvous will generate only one idle gap, and thus $V = 2$ is expected to be sufficient to achieve overlap. However, we do not observe this effect in practice.

Thus, the rendezvous protocol requires each core to have at least three processes assigned to it ($V \geq 3$) to avoid idling CPU cores unnecessarily. Note that oversubscription is only one possible solution to the problem posed here. Other options include having progress threads, handling the rendezvous on a lower level, i.e., on the NIC itself or modifying the protocol used to transmit large messages so that less steps are required.

5 CONCLUDING REMARKS

The overarching goal of this work is to better understand the idea of MPI virtualization, in which MPI processes are mapped to user-level threads and many MPI processes share the same CPU core. We find that MPI virtualization necessitates the time-measurement interface of MPI, currently consisting of only the wall-clock timer `MPI_Wtime`, to be extended with virtualization-aware timing calls: to measure time from the point-of-view of an MPI process and from the point-of-view of a CPU core. Portable MPI programs written with MPI virtualization in mind would need these timing calls to produce meaningful performance results without relying on external instrumentation. We also observe the interplay between the communication protocol and the degree of CPU oversubscription, the *virtualization ratio*: the handshake-based MPI rendezvous protocol requires at least three MPI processes to share the same CPU core in order to achieve full computation–communication overlap. We expect these insights to be applicable to any other tasking runtime system which interleaves execution of multiple tasks or aims to overlap computation with a networking protocol involving a multi-step handshake.

ACKNOWLEDGMENTS

This work has been funded by the Mont-Blanc project, grant agreement No. 671697 and the EPIGRAM-HS project, grant agreement No. 801039.

REFERENCES

[1] D. Buntinas, G. Mercier, and W. Gropp. 2006. Data Transfers between Processes in an SMP System: Performance Study and Application to MPI. In *2006 Int. Conf. on Parallel Processing*. 487–496. https://doi.org/10.1109/ICPP.2006.31

[2] Patrick Carribault, Marc Pérache, and Hervé Jourdren. 2010. Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC. In *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, Mitsuhisa Sato, Toshihiro Hanawa, Matthias S. Müller, Barbara M. Chapman, and Bronis R. de Supinski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–14.

[3] Hoang-Vu Dang, Marc Snir, and William Gropp. 2016. Towards Millions of Communicating Threads. In *Proceedings of the 23rd European MPI Users' Group Meeting* (Edinburgh, United Kingdom) *(EuroMPI 2016)*. ACM, 1–14. https://doi.org/10.1145/2966884.2966914

[4] A. Degomme, A. Legrand, G. S. Markomanolis, M. Quinson, M. Stillwell, and F. Suter. 2017. Simulating MPI Applications: The SMPI Approach. *IEEE Transactions on Parallel and Distributed Systems* 28, 8 (Aug 2017), 2387–2400. https://doi.org/10.1109/TPDS.2017.2669305

[5] Erik D. Demaine. 1997. A Threads-Only MPI Implementation for the Development of Parallel Programs. In *Proceedings of the 11th Int. Symposium on High Performance Computing Systems (HPCS'97)*. Winnipeg, Manitoba, Canada, 153–163.

[6] Christian Engelmann. 2014. Scaling to a million cores and beyond: Using light-weight simulation to understand the challenges ahead on the road to exascale. *Future Generation Computer Systems* 30 (2014), 59 – 65. https://doi.org/10.1016/j.future.2013.04.014

[7] Andrew Friedley, Greg Bronevetsky, Torsten Hoefler, and Andrew Lumsdaine. 2013. Hybrid MPI: Efficient Message Passing for Multi-core Systems. In *Proceedings of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado). ACM, New York, NY, USA, Article 18, 11 pages. https://doi.org/10.1145/2503210.2503294

[8] Filippo Gioachin, Chee Lee, Jonathan Lifflander, Yanhua Sun, and Laxmikant Kale. 2014. Tools for Debugging and Performance Analysis. In *Parallel science and engineering applications : the Charm++ approach*, Laxmikant V Kale and Abhinav Bhatele (Eds.). CRC Press, Boca Raton, Chapter 3.

[9] Brice Goglin and Stéphanie Moreaud. 2013. KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework. *J. of Parallel and Distributed Computing* 73, 2 (Feb. 2013), 176–188. https://doi.org/10.1016/j.jpdc.2012.09.016

[10] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. 2009. *Improving Performance via Mini-applications*. Technical Report SAND2009-5574. Sandia National Laboratories.

[11] T. Hoefler and A. Lumsdaine. 2008. Message progression in parallel computing - to thread or not to thread?. In *2008 IEEE Int. Conf. on Cluster Computing*. 213–222. https://doi.org/10.1109/CLUSTR.2008.4663774

[12] Chao Huang, Orion Lawlor, and L. V. Kalé. 2003. Adaptive MPI. In *Proceedings of the 16th Int. Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*. College Station, Texas, 306–322.

[13] Laxmikant V. Kalé. 2002. The Virtualization Model of Parallel Programming : Runtime Optimizations and the State of Art. In *LACSI 2002*. Albuquerque.

[14] Laxmikant V. Kale and Sanjeev Krishnan. 1993. CHARM++: A Portable Concurrent Object Oriented System Based On C++. In *Proceedings of the 8th annual conference on Object-oriented programming systems, languages, and applications - OOPSLA '93*, Vol. 53. ACM Press, New York, New York, USA, 91–108. https://doi.org/10.1145/165854.165874 arXiv:arXiv:1011.1669v3

[15] Laxmikant V. Kale and Gengbin Zheng. 2009. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications* (2009), 265–282. https://doi.org/10.1002/9780470558027.ch13

[16] Humaira Kamal and Alan Wagner. 2010. FG-MPI: Fine-grain MPI for Multicore and Clusters. In *11th IEEE Intl. Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC) held in conjunction with IPDPS-24* (Atlanta). 1–8. https://doi.org/10.1109/IPDPSW.2010.5470773

[17] S. M. Martin, M. J. Berger, and S. B. Baden. 2017. Toucan — A Translator for Communication Tolerant MPI Applications. In *2017 IEEE IPDPS*. 998–1007. https://doi.org/10.1109/IPDPS.2017.44

[18] Message Passing Interface Forum. 2015. Message-Passing Interface. http://mpi-forum.org

[19] Stas Negara, Gengbin Zheng, Kuo-Chuan Pan, Natasha Negara, Ralph E. Johnson, Laxmikant V. Kalé, and Paul M. Ricker. 2011. Automatic MPI to AMPI Program Transformation Using Photran. In *Euro-Par 2010 Parallel Processing Workshops*,

Mario R. Guarracino, Frédéric Vivien, Jesper Larsson Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 531–539.

[20] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2014. Grappa : A Latency-Tolerant Runtime for Large-Scale Irregular Applications. *Int. Workshop on Rack-Scale Computing* (2014).

[21] Marc Pérache, Patrick Carribault, and Hervé Jourdren. 2009. *MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption.* Springer Berlin Heidelberg, Berlin, Heidelberg, 94–103. https://doi.org/10.1007/978-3-642-03770-2_16

[22] Marc Pérache, Hervé Jourdren, and Raymond Namyst. 2008. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *Euro-Par 2008 – Parallel Processing*, Emilio Luque, Tomàs Margalef, and Domingo Benítez (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 78–88.

[23] S. Prakash and R. L. Bagrodia. [n.d.]. MPI-SIM: using parallel simulation to evaluate MPI programs. In *1998 Winter Simulation Conference. Proceedings (Cat. No.98CH36274)*, Vol. 1. 467–474 vol.1. https://doi.org/10.1109/WSC.1998.745023

[24] R. Riesen. 2006. A Hybrid MPI Simulator. In *2006 IEEE Int. Conf. on Cluster Computing*. 1–9. https://doi.org/10.1109/CLUSTR.2006.311852

[25] E. R. Rodrigues, P. O. A. Navaux, J. Panetta, C. L. Mendes, and L. V. Kalé. 2010. Optimizing an MPI weather forecasting model via processor virtualization. In *2010 Int. Conf. on High Performance Computing*. 1–10. https://doi.org/10.1109/HIPC.2010.5713171

[26] T. Schneider, T. Hoefler, R. Grant, B. Barrett, and R. Brightwell. 2013. Protocols for Fully Offloaded Collective Operations on Accelerated Network Adapters. In *Parallel Processing (ICPP), 2013 42nd Int. Conf. on* (Lyon, France). 593–602.

[27] Hong Tang, Kai Shen, and Tao Yang. 2000. Program Transformation and Runtime Support for Threaded MPI Execution on Shared-memory Machines. *ACM TOPLAS* 22, 4 (July 2000), 673–700. https://doi.org/10.1145/363911.363920

[28] Jerome Vienne. 2014. Benefits of cross memory attach for mpi libraries on hpc clusters. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*. 1–6.

[29] Sam White. [n.d.]. Adaptive MPI: Overview and Recent Work. ([n. d.]). Charm++ Workshop 2016; talk slides; retrieved on 12 October 2018 from http://charm.cs.illinois.edu/newPapers/16-09/AMPI.pdf.

[30] Tim S. Woodall, Galen M. Shipman, George Bosilca, Richard L. Graham, and Arthur B. Maccabe. 2006. High Performance RDMA Protocols in HPC. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Bernd Mohr, Jesper Larsson Träff, Joachim Worringen, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 76–85.

[31] G. Zheng, S. Negara, C. L. Mendes, L. V. Kale, and E. R. Rodrigues. 2011. Automatic Handling of Global Variables for Multi-threaded MPI Programs. In *2011 IEEE 17th Int. Conf. on Parallel and Distributed Systems*. 220–227. https://doi.org/10.1109/ICPADS.2011.33